Parallelization of Volume of Fluid Algorithms on Unstructured Meshes

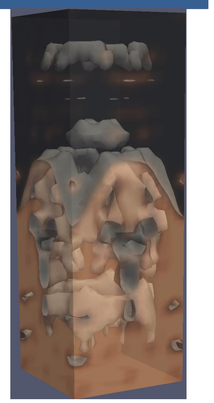
Donald Kruse[1], Alonso Navarro[2], Justin Sunu[2,3]
Mentors: Neil Carlson[4], Zach Jibben[4]

[1] Department of Mathematics and Statistics, University of New Mexico
[2] College of Sciences, San Diego State University
[3] Institute of Mathematical Sciences, Claremont Graduate University
[4] Los Alamos National Laboratory

Motivation

The development of advanced manufacturing techniques is a key contributor to US industrial competitiveness and energy efficiency. Conducting experiments on manufacturing processes can be both time consuming and costly, which calls for the use of computational simulation. Truchas is a software suite written in modern Fortran that is used to simulate a number of different manufacturing methods, such as metal casting and additive manufacturing, with the goal of obtaining crucial design information without performing physical experiments.

A key and very time consuming component of Truchas deals with the advection of interfaces separating immiscible media. In order to optimize the advection segment of the code, our research is focused on the implementation of high-level OpenMP, vectorization, and code restructuring. These changes are done on Pececillo, a mini-app of Truchas.



Material layout for the Rayleigh Taylor instability

Goals

Our overall task is improving the speed of Pececillo using three different approaches: high-level OpenMP (HLOMP), vectorization, and code restructuring.

- High-level OpenMP is implemented around computationally intensive routines.
- Vectorization is applied in regions where multiple data undergo the same math operation (SIMD).
- Code restructuring consists of reorganizing the code to remove idling time, remove repetitious tasks, remove excessive memory movement, and effectively sharing variables.

Intel Haswell and Knights Landing (KNL) architecture systems were used in this study.

Challenges

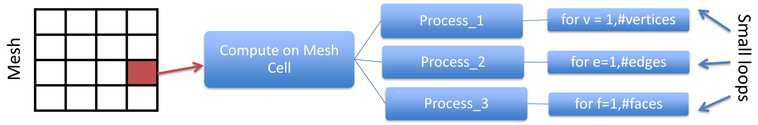
OpenMP

- High-level OpenMP requires minimizing the number of parallel regions, proper variable scoping, allocation, deallocation, etc (opening and closing parallel regions have costly overhead).
- Nested function and subroutine calls from object oriented Fortran.
- Barriers, master regions decrease threading efficiency.
- Alteration of scope when changing parallel region.

Data Dependent Algorithm

Vectorization

- Most efficient when applied to large loops with high arithmetic intensity.
- High modularity of the code causes most loops in code to be small, which suffer from slowdown if vectorized.
- Compiler favors explicit code, but the level of explicitness can be unpredictable.
- Vectorization can cause slowdowns; needs to be checked on a case by case basis.
- Unstructured grids are not easily vectorizable due to indirect addressing.



Code restructuring

- Restructuring for optimization tends to make the code messy and difficult to read.
- Code is large and well written so there are no easy places to start.
- Goes against software design best practices.

Implementations and Methods

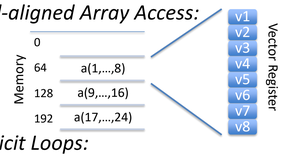
- Auto-vectorization initiated with Intel compiler flags. Additional flags are used to instruct the compiler to use AVX512 instructions and alignment of vectors.
- Many regions of the code were vectorized without any manipulation, however, there were several regions that required restructuring based on the optimization report produced from the Intel compiler.
- When restructuring code for optimization, there is a tradeoff between readability and performance.

Commonly used fixes for optimization

Indirect Accessing:

```
f(j(:))   →   F_new(i) = f(j(i))
```

Well-aligned Array Access:



Unrolling Loops:

```
do i=1,3          →   f(1)=0
  f(i)=0              f(2)=0
end do                f(3)=0
```

Explicit Loops:

```
f(:) = 1   →   do i=1,n
                 f(i)=1
               end do
```

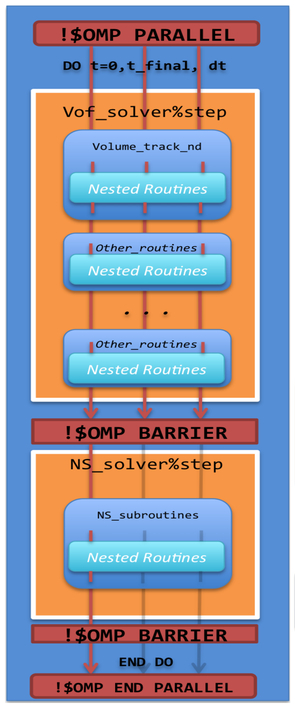
Inlining Functions:

```
f=sum(a,b)   →   f=a+b
```

Loop-carried dependencies:

```
do i=1,n                    do i=1,n
 f(:)=function(i,…)          f(:,i)=function(i,…)
 soln=soln+f(:)      →      end do
end do                      do i=1,n
                              soln=soln+f(:,i)
                            end do
```

OpenMP patterns and code restructuring



Final HLOMP pattern implemented in Pececillo

```
DO t=0,tf,dt                →   !$OMP PARALLEL
 !$OMP PARALLEL                 DO t=0,tf,dt
 !parallel computations          !parallel computations
 !$OMP END PARALLEL             END DO
END DO                          !$OMP END PARALLEL
```

Attempt to open and close a parallel region only once. Start the parallel region low and broaden for separate sections. Merge the parallel region when possible[1].

```
g(:) = gradient(f)   →   call compute_gradient(f,g)
```

Reduce memory usage by changing array functions to subroutines.

```
subroutine sub_1(f,g) ! Called in parallel region
  real(r8), intent(inout) :: f,g
  real(r8), save :: h(N)
  !! Process done on h, all threads alter same h
end subroutine sub_1
```

Use save to force OMP SHARED properties upon variables in parallel regions.

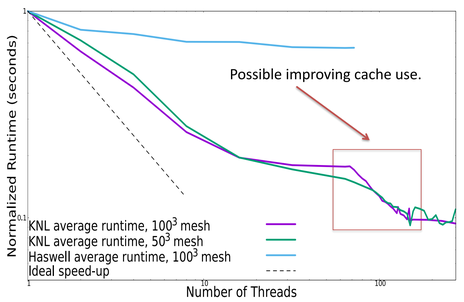
```
!$OMP END DO   →   !$OMP END DO NOWAIT
```

Finally optimize around OMP DO regions and barriers. Attempt to reduce thread spinning.

Results and Conclusions

Tests on Volume of Fluids (VOF) were run on Intel® Xeon® Processor E7-8880 v3 (Haswell) and Intel® Xeon Phi™ Processor 7250 (Knights Landing).

Strong Scaling in VOF



OpenMP

- 10.66x speedup factor with KNL with 272 threads.
- 1.5x speedup factor with Haswell on 72 threads.
- Single parallel region even with OOP code structure.
- Some barriers can't be avoided due to algorithm being heavily dependent on prior computations as well as allocations/deallocations.

- At max number of threads, KNL 17% faster than Haswell.
- Decent strong scaling occurs on the KNL up to about 10 threads only.

Vectorization

- Vectorization and code optimization gave significant speed-ups in the arithmetically intense subroutines.

Methods applied to *Gradient*	Cumulative speedup in serial (KNL)
Explicit Loops and inlining	2.03
+ Unrolling	3.31
+ Indirect addressing	3.77
+ Pre-compute data	4.00

Methods applied to *Volume*	Cumulative speedup in serial (HW)
Nested subroutine	1.2
+ Inline function	1.5
+ Force vectorization with SIMD dir	1.65

Methods in nested subroutine	Cumulative speedup in this subroutine
Indirect addressing	1.2
+ Inline functions	2.5
+ Force vectorization with SIMD dir	2.7

Future Work

- High-level OpenMP in the Navier-Stokes Solver.
- Further Optimizations around OMP DO regions and OMP BARRIERS.
- Investigate cache use on the KNL processor and the plateau after 10 threads (KNL).
- Adjust data structures for better vectorization.
- MPI + OpenMP is a natural next step for more parallization.
- Determine why scaling performance is poor on Haswell.

Acknowledgements

Many thanks to our mentors: Bob Robey, Hai Ah Nam, Kris Garrett, Doug Jacobsen, Neil Carlson, and Zach Jibben.

Support for this work was provided by the ASC Integrated Codes Program and the New Mexico Consortium.

This work was carried out under the auspices of the National Nuclear Security Administration of the us Department of Energy at Los Alamos National Laboratory supported by Contract No. DE-AC52-06NA25396.

This work was performed using the Darwin and Trinitite system at Los Alamos National Laboratory .

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Y. Zamora, J. Schoonover, and R. W. Robey. *Effective OpenMP for Extreme Scale Applications.* Los Alamos Technical Report LA-UR-17-23097, 2017.
[2] Z. Jibben. *Incompressible Multimaterial Flow in Pececillo*. Los Alamos Technical Report LA-UR-24544, 2016.

LA-UR-17-26768